2К

УДК 004.832.23+004.942

*А.В. Колчин*
Институт кибернетики имени В.М. Глушкова НАН Украины, Украина
Украина, 03680 МСП, г. Киев, просп. Академика Глушкова, 40

# Метод редукции анализируемого пространства поведения при верификации формальных моделей распределенных программных систем

*A. Kolchin*
*V.M. Glushkov Institute of cybernetics NAS of Ukraine, Ukraine*
*Ukraine, 03680 MSP, c.Kiev, Akademika Glushkova ave., 40*

# *A Method for Reduction of Analyzed Behavior Space During Verification of Formal Models of Distributed Software Systems*

*О.В. Колчин*
Інститут кібернетики імені В.М. Глушкова НАН України, Україна
Україна, м. Київ, просп. Академіка Глушкова, 40

# Метод редукції простору поведінки, що аналізується, при верифікації формальних моделей розподілених програмних систем

В основе предложенного метода лежит алгоритм отсечения избыточных по отношению к проверяемым свойствам ветвей поведения формальной модели. Факт избыточности устанавливается на основании доказательства изоморфизма на графе информационных зависимостей модели. Во многих случаях такой подход существенно сокращает эффект «комбинаторного взрыва» количества состояний.
**Ключевые слова:** верификация, редукция пространства поиска.

The core of the proposed method is an algorithm for cutting of formal model behavior branches, which are redundant with respect to verified properties. The fact of redundancy is derived basing on proof of isomorphism on the model's informational dependency graph. In many cases, such approach significantly reduces «state space combinatorial explosion» effect.
**Key words:** model checking, state space reduction.

В основі запропонованого методу лежить алгоритм відсікання надлишкових по відношенню до властивостей, що перевіряються, гілок поведінки формальної моделі. Факт надмірності встановлюється на підставі доказу ізоморфізму на графі інформаційних залежностей моделі. В багатьох випадках такий підхід істотно зменшує ефект «комбінаторного вибуху» кількості станів.
**Ключові слова:** верифікація, редукція простору пошуку.

## Введение

Комбинаторный взрыв вариантов поведения – основная проблема, с которой сталкиваются методы проверки правильности формальных моделей. Универсального решения данной проблемы не существует, однако активно создаются и развиваются специиализированные методы, позволяющие минимизировать количество состояний модели,

необходимых для проверки тех или иных свойств [1], [2]. Цели предлагаемого метода близки к целям методов частичного порядка [3], [4], которые служат для устранения избыточного интерливинга. Их суть заключается в том, чтобы при проверке свойств рассматривать не все перестановки параллельных действий в модели, а лишь некоторые из них. Существующие методы показывают высокую эффективность, когда удается синтаксически (на основании структурного описания модели) локализовать места доступа к глобальным атрибутам и каналам сообщения процессов (например, [5]). Но, к сожалению, такой подход эффективен не всегда – статический анализ значительно преувеличивает фактические информационные связи, и как следствие, существенная часть избыточных перестановок не устраняется. Более того, интерливинг часто может быть промоделирован недетерминизмом (например, когда в модели «синтаксически» всего один процесс, но его поведение представляет большой цикл недетерминированных действий). Проблема элиминации избыточных перестановок актуальна не только для верификации, отладки и тестирования, но и для повышения степени параллелизма – выявление независимых участков поведений упрощает построение адекватной распределенной архитектуры разрабатываемой системы.

Цель метода – эффективно сократить перебор вариантов поведения формальных моделей при доказательстве достижимости заданных свойств.

# Краткое описание метода

Описываемый метод развивает предложенный ранее метод динамической абстракции [6]. Главная отличительная особенность новации заключается в способе представления состояний модели. Следует отметить, что существующие Верификаторы оперируют [1] «монолитными» (т.е. представляющими собой мгновенный снимок всей информации о моделируемой системе) состояниями (иногда разделенными на части по принципу принадлежности атрибутов процессу [7], но это скорее для экономии памяти, а не для сокращения перебора), и для избегания повторных посещений осуществляют проверку вхождения текущего состояния во множество ранее пройденных. В отличие от этого, предлагаемый метод представляет поведение модели графом информационных зависимостей между значениями ее атрибутов, а для проверки повторных (точнее, избыточных) посещений используется процедура отсечения ветвей поведения, которая устанавливает факт их избыточности на основании доказательства изоморфизма уже построенной части графа и графа, который может быть построен из текущего состояния.

Для демонстрации преимущества такого подхода рассмотрим модель, состоящую из $p$ процессов, каждый из которых оперирует одним атрибутом и независимо друг от друга выполняет $n$ атомарных последовательных действий. Количество возможных трасс такой модели определяется формулой $\frac{(n \cdot p)!}{(n!)^p}$, а количество различных достижимых состояний – $(n+1)^p$, при этом граф информационных зависимостей имеет всего $(n+1) \cdot p$ вершин.

**Пример**. Пусть задана модель, в которой количество процессов p = 2, действий n = 3. Первый процесс представлен атрибутом x, второй – y, начальное состояние: x = 0, y = 0; переходы приведены в табл. 1. Свойство, необходимое для проверки на достижимость (целевое состояние Prop): $x=3 \wedge y=3$. Поток управления не структурирован (отсутствует как таковой), а последовательность выполнения переходов определяется динамически на основании значений атрибутов. Такая модель допускает 20 трасс, ведущих к целевому состоянию: ($t_1$={A0,A1,A2,B0,B1,B2}; $t_2$={A0,A1,B0,A2,B1,B2};…; $t_{20}$={B0,B1,B2,A0,A1,A2}) и 16 достижимых состояний: ({0,0},{0,1},…,{0,3},…,{3,3}).

При этом граф информационных зависимостей имеет всего 8 вершин (см. рис. 1) со значениями атрибутов (2 из которых относятся к начальному состоянию). Отметим, что для аналогичной модели из трех процессов, выполняющих по шесть действий, количество состояний превышает 300, трасс более 17 млн, тогда как граф ее зависимостей включает всего 21 вершину.

Таблица 1 – Переходы модели

| Переход | предусловие | постусловие |
|---|---|---|
| A0 | x=0 | x:=1 |
| A1 | x=1 | x:=2 |
| A2 | x=2 | x:=3 |
| B0 | y=0 | y:=1 |
| B1 | y=1 | y:=2 |
| B2 | y=2 | y:=3 |

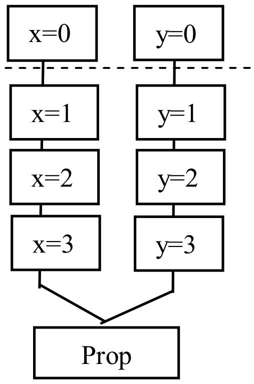



Рисунок 1 – Граф зависимости

Для данного примера выигрыш в необходимом объеме памяти очевиден; теперь нужно показать, что асимптотическая емкостная сложность хранения вершин графа зависимостей будет не больше сложности хранения «монолитных» состояний всегда. Так же будет показано, что два изоморфных графа зависимостей соответствуют одинаковым результатам проверки свойств.

Далее приведено описание формальной модели и принцип построения графа зависимости, а так же приведена аналитическая процедура прогнозирования изоморфизма и некоторые ее возможные оптимизации.

# Формальная модель

Ниже определена операционная семантика формальной модели. Пусть задано конечное множество атрибутов $A = v_1, v_2, ..., v_n$ и пусть также для каждого атрибута $v_i \in A$ определена конечная область допустимых значений $D(v_i)$.

*Определение* 1**.** Состоянием называется множество пар атрибутов и их значений (констант) вида $\{\bigcup_{i=0..|A|}(v_i = d_i) \mid v_i \in A,\ d_i \in D(v_i)\}$.

Назовем предусловием некоторую бескванторную формулу логики предикатов над атрибутами множества *A* и константами из соответствующих областей допустимых значений *D*. Назовем постусловием множество присваиваний вида *v* := *expr*(*s*), где $v \in A$, *s* – состояние, а $expr{:}s \rightarrow D(v)$ – некоторая функция над атрибутами состояния *s*.

*Определение* 2. Переходом называется тройка вида $(t,\alpha,\beta)$, где $t$ – имя перехода, $\alpha$ – его предусловие, а $\beta$ – постусловие.

Семантика переходов аналогична охраняемым командам Дейкстры [8]: если в некотором состоянии s предусловие перехода t истинно, то модель может выполнить этот переход и перейти в новое состояние s'=t(s), которое отличается от предыдущего значениями тех атрибутов, которым было выполнено присваивание новых значений в постусловии. Переходы детерминированы и выполняются атомарно за конечное время.

*Определение* 3. Формальной моделью называется семерка вида $M = <S, s_0, T, A, D, I, F>$, где $S$ – конечное множество состояний, $s_0 \in S$ – начальное состояние, $T$ – конечное множество переходов, $A$ – атрибутов, $D$ – соответствующие области допустимых значений. Интерпретация атомарных формул предусловий задана функцией $I: S\times\Pi \to \{\mathbf{T},\mathbf{F}\}$, где $\Pi$ – множество всех атомарных формул из предусловий переходов и проверяемых свойств, **T** и **F** соответственно обозначают «истина» и «ложь». $F:(s,v) \to D(v)$ – функция, вычисляющая значение атрибута $v$ на состоянии $s$.

Такая формальная модель служит удобным математическим аппаратом для описания поведения широкого спектра систем асинхронно-взаимодействующих процессов и их абстракций. Атрибуты могут быть логически разбиты на подмножества в соответствии с их принадлежностью процессам; переходы могут быть параметризированы идентификатором процесса.

*Определение* 4. Истинность формулы φ на состоянии s, записывается s|=φ, определяется индуктивно по структуре формулы φ:

s |= a ≡ I(s,a) = **T** – *если атомарный предикат* a *истинен в* s, *иначе* **F**;

s |= ¬φ ≡ s |≠ φ – *если формула* φ *не выполняется в* s;

s |= $\varphi_1 \wedge \varphi_2$ ≡ s |= $\varphi_1 \wedge$ s |= $\varphi_2$ – *если в s выполняется формула* $\varphi_1$ *и* $\varphi_2$;

s |= $\varphi_1 \vee \varphi_2$ ≡ s |= $\varphi_1 \vee$ s |= $\varphi_2$ – *если в s выполняется формула* $\varphi_1$ *или* $\varphi_2$.

Запись $s \xrightarrow{t} s'$ означает, что предусловие перехода $t$ выполняется в состоянии $s$, то есть $s \models \alpha_t$ и выполнение присваиваний постусловия $\beta_t$ преобразует $s$ в $s'$.

*Определение* 5. Трассой в $M$ из состояния $s_i$ в состояние $s_j$ называется такая последовательность состояний и переходов $s_i \xrightarrow{t_i} s_{i+1} \xrightarrow{t_{i+1}} s_{i+2} \ldots s_j$, что $s_k \in S \wedge t_k \in T$ для всех $k \in i..j$.

*Определение* 6. Состояние $s$ достижимо в модели $M$, если существует трасса, ведущая из начального состояния $s_0$ в $s$. Множество всех достижимых состояний будем обозначать Reachable($M$, $s_0$).

Под поведением модели в общем случае понимается множество ее трасс, выходящих из начального состояния.

В основном, на формальных моделях проверяют достижимость состояний, которые удовлетворяют заданным свойствам (т.н. liveness), или, наоборот, нарушают их (safety). К такой постановке задачи сводятся многие типы проверяемых свойств [2]. Для простоты описания, мы будем предполагать, что проверяемые на достижимость состояния должны удовлетворять свойствам, которые заданы в виде формулы предусловия перехода. Отметим, что это не является ограничением самого метода, в частности, метод легко расширяется для проверки свойств, выраженных в темпоральных логиках.

*Определение* 7. Задача верификации – проверить истинность $M$|=Prop, то есть найти такое состояние $s \in$ Reachable($M$, $s_0$), что $s$|=Prop, или доказать, что такого не существует.

Очевидно, что для решения такой задачи достаточно построить все множество достижимых состояний и проверить выполнимость заданного свойства на каждом из них. Однако это множество может оказаться очень большим, и как следствие, такой наивный подход окажется непригодным. Но во многих случаях для проверки свойств не обязательно строить все множество достижимых состояний. В нашем случае это может быть связано с тем, что алгоритм обнаружит искомое состояние не в последнюю очередь (и тогда его работа будет закончена), или с тем, что алгоритм при выполнении обхода не будет посещать некоторые ветви поведения модели на основании выводов о недостижимости искомого состояния в них.

# Выполнение переходов и проверка свойств

Для выполнения перехода `t` из состояния `s` в новое состояние `s'` данная работа использует процедуру `transit`; ее подробное описание приведено в [6], а модификация процедуры выполнения предусловия в [9]. Так, выполнение `transit` построит множество `R` атрибутов, значений которых (согласно лемме 1 из [9]) достаточно для вычисления истинности предусловия `t`, и (согласно лемме 4 из [6]), если переход выполним, множество `W` атрибутов, которым осуществлялось присваивание, а так же для каждого атрибута $w \in W$ множество `V[w]` атрибутов, которые входят в выражение, формирующее значение `w`. Примеры ее работы с выполнимыми переходами продемонстрированы на рис. 2, 3.

| |
|---|
| **Предусловие перехода t1 :** $\neg$(x=a+b) $\wedge$ y=0<br>**Постусловие перехода t1 :** a:=a-c; x:=0 |
| **Вход**: transit((a=2,b=0,c=1,x=1,y=0),t1)<br>**Выход**:<br>(result=**T**; s'=(a=1,b=0,c=1,x=0,y=0);<br>R={x,a,b,y};W={a,x};V[a]={a,c},V[x]=$\varnothing$ ) |

Рисунок 2 – Пример работы `transit`

| |
|---|
| **Предусловие перехода t2 :** (a>0 $\vee$ b=1)<br>**Постусловие перехода t2:** a:=a-1; c:=b |
| **Вход**: transit((a=2, b=0, c=1),t2)<br>**Выход**:<br>(result=**T**; s'=(a=1, b=0, c=0);<br>R={a};W={a,c};V[a]={a},V[c]={b} |

Рисунок 3 – Пример работы `transit`

Необходимо отметить, что в результате работы процедуры `transit` множество `R`-атрибутов будет содержать не обязательно все атрибуты, входящие в предусловие. Например, для вычисления $f_1 \wedge f_2 \wedge ... f_n$ при истинных $f_1, ..., f_{n-1}$ и ложном $f_n$, множество R будет содержать только атрибуты, входящие в $f_n$; аналогично, только атрибуты, входящие в $f_n$, будут в множестве R при вычислении $\neg(f_1 \vee f_2 \vee ... f_n)$ при ложных $f_1, ..., f_{n-1}$ и истинном $f_n$. На рис. 4 приведены примеры таких множеств, построенных для недопустимых переходов из некоторых состояний. Аналогично проверяется истинность заданного свойства (рис. 5) процедурой `property_check` [6].

| **Предусловие t1:** $\neg$(x=a+b) $\wedge$ y=0 | |
|---|---|
| **Вход – состояние q** | **Выход – множество R** |
| q={a=0,b=0,x=0,y=0} | (F; {x,a,b}) |
| q={a=0,b=0,x=1,y=1} | (F; {y}) |

Рисунок 4 – Примеры множества `R`

| **Свойство Prop:** x=0 $\vee$ $\neg$(y=0 $\vee$ z=0) | |
|---|---|
| **Вход – состояние q** | **Выход – множество R** |
| q={x=1,y=0,z=0} | (F; {x,y}) |
| q={x=1,y=1,z=0} | (F; {x,z}) |

Рисунок 5 – Примеры множества `R`

Такой способ интерпретации формул предусловий и проверяемых свойств позволяет динамически сокращать информационные связи. Пример на рис. 4 показывает, что при `y=1` значения атрибутов `x`, `a`, `b` не влияют на результат выполнения перехо-

да t1, и как следствие, будут игнорироваться. Развитием этого способа может стать «символьная» интерпретация, в результате которой в данном примере атрибуты x, a, b будут игнорироваться при ¬(y=0).

# Построение графа информационных зависимостей

Формализация информационной зависимости и алгоритмизация ее построения – актуальная тема многочисленных исследований [10-17]. Выявление независимых элементов модели (операторов, переходов) имеет широкий спектр применений – в частности, слайсинг, тестирование, отладка и т.п.

В отличие от многих существующих подходов, мы будем строить граф зависимостей значений атрибутов: его вершины (см. рис. 6) будут размечены, в частности, парами вида «атрибут=значение» (далее – «атрибутные вершины»), а направленные ребра будут отмечать факт зависимости, при этом они могут быть размечены именами соответствующих переходов. Граф будет генерироваться динамически, в процессе обхода пространства поведения модели.

Для построения графа зависимости используется адаптированный алгоритм Тарьяна поиска компонент сильной связности (КСС). Далее предполагается, что процедура transit модифицирована, так, что множества R, W, V[] будут состоять из пар вида «name, value», где name∈A – имя атрибута, value∈D(name) – его значение. Дуги и вершины графа добавляются на основании результатов (result,R,W,V[W]) выполнений процедуры transit(s,t) следующим образом: если переход выполним (result=**T**), то для каждого w∈W пара «w.name, w.value» будет соединена дугами, выходящими из всех пар «r.name, r.value», r∈R, и из пар «v.name, v.value», v∈V[w]. В случае, если переход невыполним, будет порождена специальная вершина с разметкой «t=**F**» (рис. 7), в которую будут входить дуги из всех пар «r.name, r.value», r∈R. Аналогично строятся вершины при выполнении процедуры property_check(s): в случае, если свойство, заданное для проверки, выполнимо, будет порождена специальная вершина с разметкой «Prop=**T**», иначе «Prop=**F**». Вершины вида «t=**F**», «Prop=**T**(**F**)» будем называть «синхронизирующими вершинами». Для экономии памяти и времени, такие вершины могут объединяться по принципу связи с общими атрибутными вершинами. Для того чтобы различать одинаково размеченные вершины графа, им будет сопоставляться уникальный идентификатор.

Примеры. На рис. 6, 7 показаны подграфы, построенные для выполнения перехода t1 (рис. 2, 4). Для визуализации часто удобно представлять граф двудольным, рассматривая выполнение переходов и свойств как специальные вершины (см. соответственно рис. 8, 9).

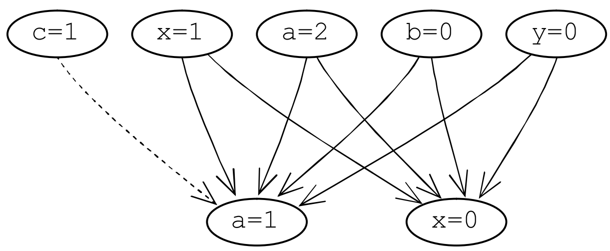



Рисунок 6 – Подграф для выполнимого перехода на рис. 2

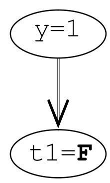



Рисунок 7 – Подграф для невыполнимого перехода на рис. 4

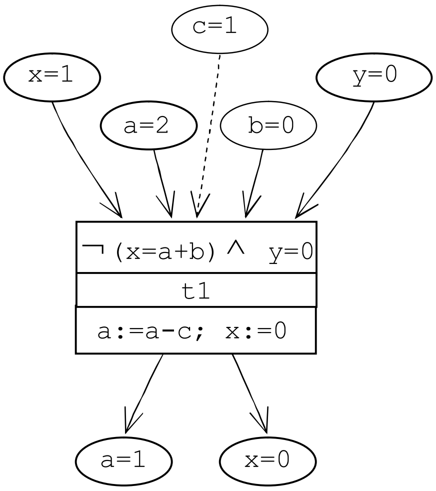



Рисунок 8 – Двудольный подграф для перехода на рис. 2

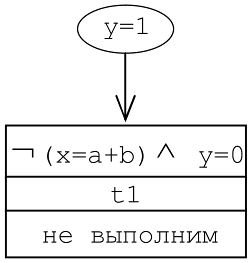



Рисунок 9 – Двудольный подграф для перехода на рис. 4

Стоит отметить, что дуга от вершины «c=1» изображена пунктирной линией, так как она входит во множество V[a], но не в R (интуитивно – она представляет поток данных, а не управления). Такое различие позволит производить дополнительное отсечение ветвей в графе зависимости: если из вершины «a=1» не будет выходить ни одной дуги, то вершину «c=1» можно игнорировать (при условии отсутствия других выходящих из нее дуг), а следовательно, не различать значения атрибута c при проверке повторности прохождения состояния.

*Определение* 8**.** Графом зависимости называется размеченный ориентированный ациклический граф, разметки вершин которого представляют собой пары вида «атрибут=значение», «transition=**F**», «Prop=**T**(**F**)», а дуги отмечают факт зависимости. Граф, порождаемый состоянием s, будет обозначаться SubDG(s).

Далее предполагается, что информационная зависимость модели представлена графом, удовлетворяющим определению 8 (далее для краткости D-граф).

*Определение* 9**.** Два D-графа $G_1$ и $G_2$ называются изоморфными, если $G_1$ после перенумерации идентификаторов вершин становится равным $G_2$, то есть множества их вершин и дуг совпадает, в $G_1$ есть дуга, соединяющая вершины с идентификаторами *i* и *j* тогда и только тогда, когда в $G_2$ есть дуга, соединяющая вершины *i* и *j*.

Полные формальные доказательства свойств алгоритма описываемого метода потребуют описания алгоритма проверки свойств модели методом динамической абстракции [6], так как предполагается, что вызов некоторых процедур (в частности, property_check) будет производиться (соответствующим образом модифицированной) процедурой traverse. Поэтому здесь и далее приводятся схемы доказательств со ссылками на результаты [6].

**Теорема 1.** При завершении обхода компоненты сильной связности SCC поведения модели для ее корня $ROOT_{scc}$ выполняются свойства:

(1) $\exists s(s \in SCC \wedge s \models Prop) \Leftrightarrow$ «Prop=**T**»$\in SubDG(ROOT_{scc})$,

(2) $(s \in SCC \Rightarrow s \not\models Prop) \Leftrightarrow$ «Prop=**T**»$\notin SubDG(ROOT_{scc})$.

*Схема доказательства.* Вначале нужно показать, что процедура traverse будет вызвана для каждого состояния из SCC. Далее справедливость первого утверждения следует из того, что вызов процедуры property_check осуществляется в проце-

дуре `traverse` для каждого нового состояния, следовательно, если `s|=Prop`, то вершина «`Prop=`**`T`**» построена при выполнении `property_check(s)`, и значит, входит в подграф `SubDG(ROOT`$_{scc}$`)`. Аналогично доказывается второе утверждение.

Следует отметить, что при поиске циклов метод может редуцировать множество состояний КСС, которые будут параметром процедуры `traverse`, но такая редукция сохранит справедливость теоремы 1. Полная процедура поиска циклов и КСС сама по себе заслуживает отдельного рассмотрения, ее полное описание выходит за рамки данной статьи.

# Алгоритм отсечения избыточных ветвей поведения

При обходе пространства поведения формальной модели актуальна задача проверки вхождения текущего состояния во множество ранее пройденных (так называемых visited states). Это позволяет избежать повторных посещений состояний и определять цикличность поведения. Для этих целей существующие методы эффективно используют хеширование состояний (например, [18]). В этом подходе заключается суть различия предлагаемого метода от существующих. Предлагаемый в этой работе метод представляет пройденную часть пространства поведения модели в виде графа информационных связей, а установление повторности посещения состояния сводится к доказательству изоморфизма уже построенной части графа и графа, который может быть построен из текущего состояния. Справедливость последнего утверждения следует из того, что, согласно теореме 1, выполнимость проверяемых свойств консервативна относительно D-графа, а значит, при обходе пространства поведения модели можно ослабить отношение эквивалентности состояний (трассовой эквивалентности), и проверять одинаковость поведения с точностью до изоморфизма D-графов.

Пусть $G=(V, E)$ – D-граф. Если $(a, b)\in E$, будем писать $a \Downarrow_G b$. Через $\Downarrow_G^+$ будем обозначать транзитивное замыкание отношения $\Downarrow_G$. Будем писать $a \Updownarrow_G b$, если существует такая вершина $x$, что $a \Downarrow_G x \wedge b \Downarrow_G x \wedge \neg(a \Downarrow_G^+ b) \wedge \neg(b \Downarrow_G^+ a)$. Через $\Updownarrow_G^+$ будем обозначать транзитивное замыкание отношения $\Updownarrow_G$. Согласно методу построения динамических абстракций [6], для установления факта повторности посещения состояния (достаточного для трассовой эквивалентности), достаточно сравнивать состояния с точностью до множества R-атрибутов (интуитивно, это множество пар вида `атрибут = значение`, к которым осуществлялся «доступ на чтение»). В [6] предполагалось, что пройденные состояния хранятся как «монолитные». Предлагаемое усовершенствование заключается в том, что R-множество будет каждый раз строиться динамически по алгоритму транзитивного замыкания отношения $\Updownarrow_G$ на D-графе. Иными словами, для обнаружения полного множества R-атрибутов, соответствующего текущему состоянию $S$, предлагаемый метод повторит прохождение соответствующих участков поведения модели, то есть работа алгоритма сведется к проверке ложности утверждения $a \Updownarrow_G^+ x \Rightarrow x \notin S$ для каждой вершины $a \in$ `local_r_set`. Такая процедура, конечно, значительно медленнее существующих эффективных процедур поиска состояния в базе пройденных по хеш-функции. Но с другой стороны, за счет того, что повторное прохождение осуществляется только по D-графу, такой метод позволяет эффективно устранять избыточные перестановки при обходе поведения модели.

Далее предполагается, что существует процедура обхода пространства поведения модели аналогичная процедуре `traverse` [6]. Вершины для текущего состояния представлены «монолитно» структурой `current`. Также предполагается, что построен-

ные КСС представлены D-графом G, в котором «изоморфные» вершины (то есть вершины с одинаковой разметкой, достижимые подграфы из которых изоморфны) связаны специальными ссылками `visited_reference`. Структура данных для хранения информации о вершинах графа такова: {`name`, `value`, `ingoing`, `outgoing`, `idem`, `visit`}, где `ingoing` и `outgoing` обозначают множества вершин, инцидентных соответственно входящим и исходящим дугам; `idem`/`visit` используются для хранения связей с «изоморфными» вершинами. Хранилище `nodes_storage` индексирует граф по разметке его вершин.

Процедура `check_visited:S`$\to${**T**,**F**} используется для обнаружения избыточности посещения состояний по отношению к КСС, которые найдены к моменту ее вызова. Предполагается, что вызов процедуры `check_visited` будет производиться процедурой `traverse` для каждого вновь построенного состояния.

# Алгоритм отсечения избыточных ветвей

*Вход.* Множество `local_r_set` пар вида (`name`, `value`), построенное из множеств R выполнения всех переходов и проверяемых свойств на данном состоянии.
*Выход.* Резолюция ‘изоморфен’, если подграф, который можно построить из текущего состояния, изоморфен подграфу, построенному ранее (следовательно, анализ текущей ветви поведения можно не продолжать); в противном случае – резолюция ‘не изоморфен’. Так же обновляются связи `visited_reference`.

```
    procedure check_visited(local_r_set)local(r, nd, o, flag)
    begin
1.      for r ∈ local_r_set do
        begin
2.          for nd ∈ nodes_storage(r.name, r.value) do
            begin
3.              clear_marks();
4.              set_mark(nd, in_progress);
5.              flag ← T;
6.              for o ∈ nd->outgoing do
                begin
7.                  flag ← check_one_node(o);
8.                  if(flag = F)then do
9.                      break  // переход к следующему элементу из nodes_storage
                end
10.             if(flag = T)then do
                    break  // переход к следующему элементу из local_r_set
            end
11.         if(flag = F)then do
12.             return ‘не изоморфен’
        end
13.     return ‘изоморфен’
    end
```

Процедура `check_one_node` проверяет изоморфизм относительно одной вершины.

```
    procedure check_one_node(nd)local(i, o, v, flag)
    begin
14.     set_mark(nd, in_progress);
15.     for i ∈ nd->ingoing do
        begin
16.         if(get_mark(i) = in_progress))then do
```

```
17.             continue;   // переход к следующему элементу из nd->ingoing
18.         flag ← T;
19.         if(i.value = current[i.name].value ∧ i ≠ current[i.name])then do
            begin
20.             current[i.name]->idem ← i;
21.             i->visit ← i->visit ∪ current[i.name]
            end
22.         if(i.value ≠ current[i.name].value)then do
23.             flag ← check_one_node(i);
24.         if(flag = F)then do
25.             for v ∈ nd->visit do
                begin
26.                 flag ← check_one_node(v);
27.                 if(flag = T)then do
28.                     break
                end
29.         if(flag = F)then do
            begin
30.             unset_visited_references(nd);
31.             return F
            end
        end
32.     if(nd->idem ≠ ∅)then do
33.         nd ← nd->idem;
34.     for o ∈ nd->outgoing do
        begin
35.         flag ← check_one_node(o);
36.         if(flag = F)then do
            begin
37.             unset_visited_references(nd);
38.             return F
            end
        end
39.     return T
    end       □
```

Процедуры clear_marks, set_mark, get_mark соответственно сбрасывают, отмечают и проверяют отметку in_progress вершин графа для того, чтобы избежать зацикливания. Так как связи «потенциально изоморфных» вершин графа устанавливаются в процессе его обхода, то в случае негативного результата нужно их удалить. Это обеспечивается процедурой unset_visited_reference.

**Теорема 2.** Алгоритм закончит работу за время O(|SubDG(current)| · |A|).

*Доказательство.* Согласно строкам 3, 4, 14, 16, 17 разметка in_progress гарантирует, что каждая вершина построенной части D-графа будет посещена не более одного раза для каждого элемента из local_r_set.

Необходимо отметить, что время можно значительно сократить путем использования техники специальной разметки, о которой будет сказано позже.

**Теорема 3.** Процедура check_visited правильно отвечает на вопрос «изоморфен ли подграф, который можно построить из текущего состояния, подграфу, построенному ранее?».

*Схема доказательства.* По индукции. Доказательство основывается на том факте, что, будучи полностью построенной, КСС уже никогда не изменится: внутри нее не добавятся ни вершины, ни дуги, следовательно, D-граф, соответствующий КСС так же не будет изменяться. В [6] доказано, что состояния достаточно сравнивать с точностью до множества R. Осталось доказать, что алгоритм построит его (множество вершин, удовлетворяющих условию в строке 19) полностью. Для этого нужно показать, что алго-

ритм правильно вычисляет транзитивное замыкание отношения $\Updownarrow_G$ для вершин множества `local_r_set`. Для вершин инцидентных по входящим дугам, это свойство выполняется согласно строкам 15, 22, 23; по выходящим соответственно 6, 7 а так же 34, 35. Строки 24 – 28, 32, 33 аналогичным образом анализируют вершины, изоморфизм которых был доказан ранее.

*Следствие.* В случае позитивного ответа ветвь поведения из текущего состояния можно не рассматривать, так как проверка в ней выполнения заданных свойств, согласно теореме 1, избыточна.

**Теорема 4.** Если состояние `S` встретилось повторно, то результатом работы процедуры `check_visited(S)` будет резолюция ‘изоморфен’.

*Схема доказательства.* От противного. Пусть результатом будет ‘не изоморфен’. Тогда, с одной стороны (согласно строкам 2, 6, 7, 34, 35), это будет означать, что существует некоторая вершина «na, nv», достижимая из $S_{prev}$. С другой, это означает (согласно строкам 15, 23), что существует путь $v_0$..«na,nv» на D-графе, не включающий ни одной вершины из $S_{prev}$ (иначе, согласно строкам 19, 22, 23, алгоритм нашел бы ее), причем $v_0$ не имеет входящих дуг, так как D-граф не имеет контуров по построению. Этот путь показывает, что вершина «na,nv» недостижима из $S_{prev}$, что приводит к противоречию.

**Теорема 5.** Пусть `|V|` – количество атрибутных вершин D-графа, `|A|` – количество атрибутов модели, `|S|` – количество достижимых «монолитных» состояний. Тогда $|S|\cdot|A| \geq |V|$.

*Доказательство.* В худшем случае каждое состояние из множества S будет представлено |A| вершинами, то есть, когда множество R содержит все атрибуты модели для каждого состояния. Тот факт, что граф не будет включать больше вершин, чем того потребует множество состояний S, следует из теоремы 4.

Из теоремы 5 можно сделать вывод о затрачиваемой памяти. Очевидно, что в случае представления поведения графом, нужны дополнительные затраты памяти для хранения дуг и другой вспомогательной информации. Так же нужна память для хранения синхронизирующих вершин (вида «`transition=`**`F`**», «`Prop=`**`T(F)`**»). Однако определяющей роли этот факт не играет: таких вершин будет максимум столько же сколько и атрибутных, следовательно, в худшем случае $m_g=k\cdot m_s$, где $m_g$ – память для хранения графа, $m_s$ – для хранения «монолитных» состояний, `k` – маленькая константа. На практике, каждый переход модели изменяет не более 3 – 4 атрибутов модели, соответственно $m_g \ll m_s$.

# Разметка графа

Для повышения эффективности процедуры `check_visited` разработан алгоритм специальной разметки D-графа. Полное его описание выходит за рамки данной статьи; здесь мы ограничимся кратким описанием. Например, если, однажды проанализировав подграф G, и установив, что он не будет изоморфным (по отношению к подграфу, который порождается текущим состоянием), его вершины будут помечены красным цветом. Вершина, имеющая красный цвет, не нуждается в обходе подграфа, достижимого из нее – отрицательный ответ в таком случае будет выдан мгновенно. Причем помимо своей разметки каждая вершина будет иметь информацию о «причине» – указатель на ту смежную вершину, без изменения разметки которой нет смысла менять свою разметку. Аналогично, зеленый цвет вершины будет означать, что подграф, достижимый из нее, уже проанализирован, что позволяет выдать положительный ответ сразу.

Для поддержания актуальности разметки необходима процедура ее обновления в соответствии с изменениями значений атрибутов модели. Так, если цвет вершины был красным или зеленым и произошло изменение, влияющее на ее разметку, то цвет становится белым, означая, что для установления изоморфности необходим дальнейший анализ вершин подграфа, достижимого из данной вершины. Такая техника позволяет существенно повысить производительность новой процедуры проверки пройденных состояний.

## Статическое отсечение избыточных зависимостей

Метод можно усовершенствовать путем дополнительного отсечения ветвей в D-графе. Суть статического отсечения заключается в распознании фактической независимости некоторых вершин на основании анализа переходов модели до начала построения графа.

**Пример.** Рассмотрим модель, переходы которой представлены в табл. 2. Анализируя описание переходов, можно сделать вывод, что вершина `control_flow=cf2` будет зависеть только от вершины `control_flow=cf1`, таким образом, остальные ее связи (с вершинами `a=0`, `b=0`, `c=0`, …) в процессе построения графа можно будет игнорировать, при этом граф гарантированно останется консервативным относительно проверяемых свойств.

Таблица 2 – Переходы модели

| переход | предусловие | постусловие |
|---|---|---|
| T1 | control_flow=cf1 ∧ a=0 ∧ b=0 | control_flow := cf2; a:=1 |
| T2 | control_flow=cf1 ∧ c=0 | control_flow := cf2; c := 0; b:=0 |
| … | | |
| Tn | control_flow=cf1 ∧ … | control_flow := cf2;… |

Так же для отсечения ветвей можно адаптировать существующие методы, уточняющие информационные зависимости [10-13].

## Динамическое отсечение избыточных зависимостей

Динамический подход заключается в последовательном выполнении процедуры стягивания дуг, ведущих к висячим атрибутным вершинам. Это вершины, к которым не было «доступа на чтение», и как следствие, они не попали во множество R. В итоге отсекается весь подграф, корнем которого служит вершина, из которой нет достижимых синхронизирующих вершин. Процедура выполняется при завершении обхода компоненты сильной связности и, очевидно, отсекает только избыточные по отношению к проверяемым свойствам ветви D-графа. Помимо сокращения пространства поиска, предложенный подход способен обнаружить потенциальные недостатки: выявление таких путей свидетельствует о «лишних» изменениях значений атрибутов, что может быть источником неэффективности вычислений или местом утечки памяти в разрабатываемой системе.

## Выводы

Предложен новый метод редукции пространства поиска при решении задачи достижимости на формальной модели системы переходов. В основе метода лежит оригинальный способ представления поведения модели – в отличие от традиционных методов проверки модели, запоминаются не индивидуальные состояния модели (включающие

полную информацию о значениях всех ее атрибутов), а граф зависимостей между значениями ее атрибутов. Показано, что такой граф может быть изоморфным для довольно большого множества различных поведений модели (трасс). Граф зависимостей задает классы эквивалентности поведения с точки зрения проверяемых свойств, что позволяет не рассматривать перестановки действий модели принадлежащие одному классу, таким образом, значительная часть ветвей поведения отсекается при верификации модели, что сокращает эффект «комбинаторного взрыва». Предложен алгоритм прогнозирования изоморфизма и некоторые его оптимизации – техника разметки вершин графа зависимостей, а так же статическая и динамическая техники отсечения ветвей в нем.

На практике применение метода позволило получить существенное сокращение пространства поиска за счет устранения избыточного интерливинга и недетерминизма. Так же метод используется для отладки поведения формальных моделей и анализа причин дефектов в них [17]. Метод совместим с существующими методами предикатной абстракции [19], что делает его применимым для символьной верификации.

# Литература

1. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем / Карпов Ю.Г. – БХВ-Петербург, 2010. – 552 С.
2. Jhala R. Software model checking / R. Jhala, R. Majumdar // ACM Comput. Surv. – 2009.— Vol.41(4). – 54 P.
3. Peled D. Partial order reduction: linear and branching temporal logics and process algebras / D. Peled // In proc. of the workshop on Partial order methods in verification. – NY, USA : AMS Press, Inc. – 1997. – P. 233-257.
4. Godefroid P. Partial-order methods for the verification of concurrent systems – an approach to the state–explosion problem / P. Godefroid // LNCS, Springer-Verlag. – 1996. – Vol. 1032. –143 P.
5. Holzmann G. An improvement in formal verification / G. Holzmann, D. Peled // FORTE Conf. – 1994. – P. 197-211.
6. Колчин А.В. Автоматический метод динамического построения абстракций состояний формальной модели // Кибернетика и системный анализ. – 2010. – № 4. – С. 70-90.
7. Holzmann G. The model checker SPIN / G. Holzmann // IEEE transactions on software engineering. – 1997. – Vol. 23, № 5. – P. 279-295.
8. Dijkstra E. Guarded commands, nondeterminacy and formal derivation of programs / E. Dijkstra // Communications of the ACM. – 1975. – Vol.18, № 8. – P. 453-457.
9. Колчин А.В. Оптимизация проверки выполнимости переходов при верификации формальных моделей / А.В. Колчин // Проблемы программирования. – 2012. – № 2-3. – С. 201-210.
10. Podgurski A. Formal model of program dependencies and its implications for software testing, debugging and maintenance / A. Podgurski, A.A. Clarke // IEEE Trans. Software Eng. – 1990. – Vol. 16(9). – P. 965-979.
11. Ottenstain K.J. The program dependence graph in a software development environment / K.J. Ottenstain, L.M. Ottenstain // In proc. of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments. –1984. – V. 19(5). – P.177-184.
12. Cortesi A. Dependence Condition Graph for Semantics-based Abstract Program Slicing / A.Cortesi, R. Halder // In Proc. of the 10th Workshop on Language Descriptions, Tools and Applications. – 2010. – № 4. – 9 P.
13. Tip F. A survey of program slicing techniques / F. Tip // Technical report. Centre for Mathematics and Computer Science. – Amsterdam : The Netherlands, 1994. – 65 P.
14. Wilander J. Modeling and visualizing security properties of code using dependence graphs / J. Wilander // In Proc. of the 5th Conference on Software Engineering Research and Practice in Sweden. – 2007. – P. 65-74.
15. Krinke J. Program Slicing / J. Krinke // In Handbook of software engineering and knowledge engineering. – Vol. 3. – 2005. – P. 307-332.
16. Ilan Beer. Explaining Counterexamples Using Causality / Ilan Beer, Shoham Ben-David and oth. // Formal Methods in System Design. – 2012. – № 40. – P. 20-40.
17. Колчин А.В. Интерактивная система для анализа поведения формальных моделей программных систем / А.В. Колчин, Р.В. Четвертак // Искусственный интеллект. –2012. – № 4. – С. 330-341.
18. Holzmann G. An analysis of bitstate hashing / G. Holzmann // Formal Methods in Systems Design. – 1998. – P.301-314.
19. Clarke E. Counterexample-guided abstraction refinement for symbolic model checking / [Clarke E., Grumberg O., Jha S., and oth.] // Journal of the ACM. –2003. – Vol. 50, № 5. – P. 752-794.

## Literatura

1. Karpov Y.G. Model Checking. – BHV-Petersburg. – 2010. – 552 P.
2. Jhala R., Majumdar R. Software model checking // ACM Comput. Surv. – Vol.41(4). – 2009. – 54 P.
3. Peled D. Partial order reduction: linear and branching temporal logics and process algebras // In proc. of the workshop on Partial order methods in verification. NY, USA: AMS Press, Inc. – 1997. – P. 233–257.
4. Godefroid P. Partial-order methods for the verification of concurrent systems – an approach to the state–explosion problem // LNCS, Springer-Verlag. – 1996. – Vol. 1032. –143 P.
5. Holzmann G., Peled D. An improvement in formal verification // FORTE Conf. – 1994. – P. 197-211.
6. Kolchin A.V. An automatic method for the dynamic construction of abstractions of states of a formal model // Cybernetics and system analysis. – 2010. – № 4. – P. 70-90.
7. Holzmann G. The model checker SPIN // IEEE transactions. – 1997. – Vol. 23. – N 5. – P. 279–295.
8. Dijkstra E. Guarded commands, nondeterminacy and formal derivation of programs // Communications of the ACM. – 1975. – Vol.18. – N 8. – P. 453–457.
9. Kolchin A.V. Enabled transitions detection optimization in formal model verification // Problems in programming. – 2012. – №2–3. – P. 201–210.
10. Podgurski A., Clarke A.A. Formal model of program dependencies and its implications for software testing, debugging and maintenance // IEEE Trans. Software Eng. – 1990. – Vol. 16(9). – P. 965-979.
11. Ottenstain K.J., Ottenstain L.M. The program dependence graph in a software development environment // In proc. of the ACM SIGSOFT/SIGPLAN software engineering symposium on practial software development environments. –1984. –V.19(5). –P.177–184.
12. Cortesi A., Halder R. Dependence Condition Graph for Semantics-based Abstract Program Slicing // In Proc. of the 10th Workshop on Language Descriptions, Tools and Applications. – 2010. – № 4. – 9 P.
13. Tip F. A survey of program slicing techniques // Technical report. Centre for Mathematics and Computer Science, Amsterdam, The Netherlands. –1994. – 65 P.
14. Wilander J. Modeling and visualizing security properties of code using dependence graphs // In Proc. of the 5th Conference on Software Engineering Research and Practice in Sweden. – 2007. – P. 65–74.
15. Krinke J. Program Slicing // In Handbook of software engineering and knowledge engineering. – 2005. – Vol. 3.– P. 307–332.
16. Ilan Beer, Shoham Ben-David and oth. Explaining Counterexamples Using Causality // Formal Methods in System Design. – 2012. – №40. – P. 20-40.
17. Kolchin A., Chetvertak R. An interactive system for analysis of formal models behavior. –2012. – № 4. – P. 330-341.
18. Holzmann G. An analysis of bitstate hashing // Formal Methods in Systems Design. – 1998. – P.301-314.
19. Clarke E., Grumberg O., Jha S., and oth. Counterexample-guided abstraction refinement for symbolic model checking // Journal of the ACM. –2003. – Vol. 50. – № 5. – P. 752-794.

***RESUME***

***A. Kolchin***

### *A Method for Reduction of Analyzed Behavior Space During Verification of Formal Models of Distributed Software Systems*

A new state-space reduction method for reachability checking on formal model of transition system is proposed. The basis of the method is a technique of model behavior form representation – in opposite to existing model checking methods, where model's individual states (values of all attributes) are stored, the method stores a dependency graph between values of model's attributes. It is shown, that dependency graph may be isomorphic for a great set of different model behaviors (traces). From the verified properties point of view, the dependency graph specifies equivalence classes of model behavior. This allows to drop permutations of model actions, which belongs to the same class, and thus, many behavior branches are cut-off during model verification, as a result, 'combinatorial explosion' effect is reduced. An algorithm for prediction of isomorphism is proposed together with some optimizations – graph vertices markup technique, static and dynamic techniques of branches cut-offs.

In practice, application of the method shows significant state-space reduction due to avoiding of the exploration of redundant interleavings and non-determinisms. Also method is applicable for model debugging and analysis of defects causes. The method is compatible with predicate abstraction techniques; this fact makes the method applicable for symbolic verification.

*Статья поступила в редакцию 16.04.2013.*